Licence Sciences et Technologie **UGA**
L2 Informatique **UE INF404**

Examen du 20 mai 2019

Durée : 2h – Une feuille A4 recto/verso autorisé – Barême indicatif.

Introduction

On s'intéresse à un langage $\mathcal{L}$ permettant de décrire et afficher un réseau d'opérateurs n-aires. On considèrera ici deux opérateurs notés `prod` et `sigma`, représentant respectivement le produit et la somme de leurs entrées. La description d'un réseau construit avec ces opérateurs consiste en :

— une **liste de déclarations** d'entrées prédéfinies, représentées par des lettres (minuscules) de l'alphabet ;
— une **expression** indiquant comment sont connectés les entrées et les opérateurs entre-eux.

On donne ci-dessous un exemple de description et la représentation graphique correspondante (Fig 1) :

```
x, y, z, t ;
sigma(x, sigma(y, z, prod (x, t)), prod (prod(z, t), t))
```

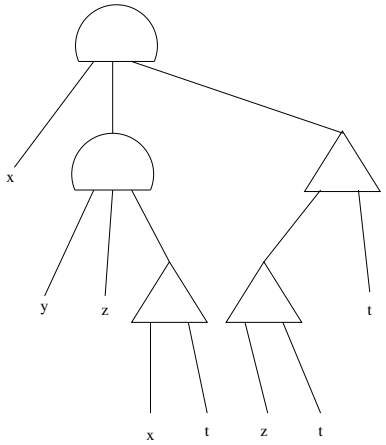

FIGURE 1 – un réseau d'opérateur

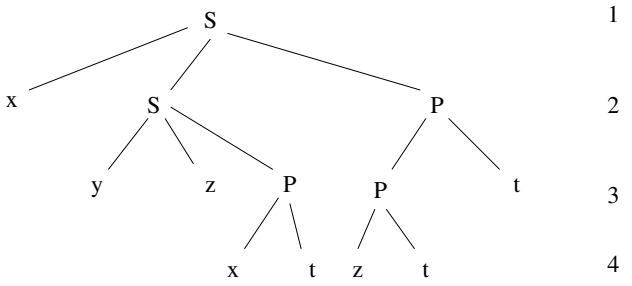

FIGURE 2 – l'arbre abstrait associé

On suppose dans la suite que chaque opérateur ne peut admettre que 6 entrées au maximum.

L'objectif de ce travail est d'écrire un programme sur le même modèle que ce qui a été fait en projet. Ce programme prendra donc en entrée une description lue au clavier et effectuera :

— une analyse lexicale : question 1 ;
— une analyse syntaxique, avec construction d'une table des symboles et d'un arbre abstrait : questions 2 et 3 ;
— une vérification de la cohérence du réseau : question 4 ;
— la construction d'une forme intermédiaire pour préparer l'affichage graphique : question 5 ;
— et enfin l'affichage du réseau dans une fenêtre graphique : question 6.

Seules les questions 2 et 3 sont liées entre-elles, et peuvent être résolues simultanément si vous le souhaitez. Les autres questions sont indépendantes et peuvent être traitées dans n'importe quel ordre.

Question 1 : analyse lexicale [2 points]

Les lexèmes du langage sont les suivants :

- `LETTRE`, représente une lettre minuscule de l'alphabet
- `SIGMA`, représente l'opérateur noté `"sigma"`
- `PROD`, représente l'opérateur noté `"prod"`
- `PARO`, représente le caractère `(`
- `PARF`, représente le caractère `)`
- `VIRG`, représente le caractère `,`
- `PVIRG`, représente le caractère `;`
- `FIN_SEQ`, représente la fin de la séquence lue au clavier
- `ERREUR`, représente une erreur lexicale

Dessinez un automate de reconnaissance de ces lexèmes. Cet automate lit en entrée une séquence de caractères et il atteint un état final lorsqu'un lexème a été reconnu. Les transitions seront étiquetées uniquement par le caractère courant. Les caractères séparateurs sont l'espace ou le "fin-de-ligne", et on ne fera pas apparaitre le traitement des erreurs lexicales.

Question 2 : analyse syntaxique [4 points]

La grammaire (sous forme LL(1)) du langage $\mathcal{L}$ est la suivante :

$$
\begin{aligned}
\textit{desc} &\rightarrow \textit{liste\_decl}\ PVIRG\ \textit{expression}\ FIN\_SEQ \\
\textit{liste\_decl} &\rightarrow LETTRE\ \textit{suite\_liste\_decl} \\
\textit{suite\_liste\_decl} &\rightarrow \varepsilon \mid VIRG\ LETTRE\ \textit{suite\_liste\_decl} \\
\textit{expression} &\rightarrow SIGMA\ PARO\ \textit{liste\_exp}\ PARF \mid PROD\ PARO\ \textit{liste\_exp}\ PARF \mid LETTRE \\
\textit{liste\_exp} &\rightarrow \textit{expression}\ \textit{suite\_liste\_exp} \\
\textit{suite\_liste\_exp} &\rightarrow \varepsilon \mid VIRG\ \textit{expression}\ \textit{suite\_liste\_exp}
\end{aligned}
$$

En utilisant le type `Lexeme` fournie en Annexe A et les primitives du module `analyse_lexicale` utilisées en projet (`Demarrer`, `Avancer`, `Lexeme_Courant`, etc) écrivez le corps de la procédure `analyser` spécifiée ci-dessous. Vous pouvez écrire des procédures auxilliaires (comme `rec_desc`, `rec_liste_decl`, etc.).

```
void analyser() ;
// e.i. : indifferent
// e.f. : une sequence de lexemes a ete lue, une Erreur
//        est levee si elle ne respecte pas la grammaire du langage L
```

Question 3 : construction d'une table des symboles et d'un arbre abstrait [4 points]

(la réponse à cette question peut être groupée avec celle de la question 2)

Conformément à la grammaire fournie, une description de réseau est constituée de deux parties : une **liste de déclarations** et une **expression**. Complétez le code C de la procédure `analyser` (question 2) pour qu'elle fournisse en paramètre résultat :

1. Une *table des symboles*, produite à partir de la **liste des déclarations**, c'est-à-dire un tableau contenant l'ensemble des entrées prédéfinies déclarées dans la description analysée ;
2. l'*arbre abstrait* correspondant à l'**expression** lue ; Il s'agira d'un arbre n-aire, chaque noeud ayant entre 0 et 6 fils. On donne sur la Figure 2 l'arbre abstrait correspondant à la description de réseau donnée en introduction (en notant P pour `prod` et S pour `sigma`).

Vous utiliserez les types et primitives de construction [1] définies en Annexe B et C. La nouvelle spécification de la procédure `analyser` est donc la suivante :

```
void analyser (TabSymb *T ; Ast *A) ;
// e.i. : indifferent
// e.f. : une sequence de lexemes a ete lue, une Erreur est levee si elle ne respecte
//        pas la grammaire du langage L, sinon *T et *A contiennent respectivement la table des
//        symboles et l'arbre abstrait de la description lue.
```

Question 4 : Vérifier la cohérence du réseau [3 points]

Il s'agit ici d'écrire une fonction qui vérifie que, sur un réseau, **toutes les entrées déclarées sont utilisées**, et, **seules les entrées déclarées sont utilisées**. On pourra pour cela utiliser (**sans les écrire**) les primitives de gestion d'ensemble fournies en Annexe D.

```
int verifier (TabSymb T ; Ast A) ;
// e.i. : T et A sont les tables de symboles et AST d'un reseau d'operateurs
// e.f. : verifier(T,A) renvoie 1 ssi toutes les entrees declarees dans T sont utilisees dans A
//        et rien que les entrees declarees dans T sont utilisees dans A.
```

Question 5 : Produire une forme intermédiaire pour l'affichage [4 points]

Pour faciliter l'affichage du réseau dans une fenêtre graphique il est préférable de produire tout d'abord une *forme intermédiaire* à partir de l'arbre abstrait. Cette forme intermédiaire, notée `L` dans la suite, contient les séquence de noeuds associés à chacun des **niveaux** de l'arbre abstrait [2]. Sur l'exemple donné en introduction, la forme intermédiaire à produire contiendrait donc :

niveau 0 : la séqence [sigma] (l'opérateur associé à la racine) ;

niveau 1 : la séquence [x, sigma, prod]

niveau 2 : la séquence [y, z, prod, prod, t]

niveau 3 : la séquence [x, t, z, t]

Ecrivez le code de la procédure `GenFormInter` suivante :

```
// NMAXNIV et NMAXNOEUDS sont des constantes
typedef struct {
  Noeud liste[NMAXNIV][NMAXNOEUDS] ; // les noeuds associes a chaque niveau
  int nb_noeuds_niv[NMAXNIV] ; // le nombre de noeuds par niveaux
}

void GenFormInter (Ast A ; listeNiveaux *L) ;
// etat initial : A est l'arbre abstrait d'un reseau d'operateurs
// etat final : L contient la liste des noeuds par niveau dans A,
//    L.liste[0][0] est la racine
//    L.liste[1] est la sequence des noeuds de niveau 1  (ordonnee de gauche a droite)
//    L.nb_noeud_niv[1] est le nombre de noeuds de niveau 1
//    etc.
```

Indications : On pourra utiliser des procédures annexes, mais il faut les spécifier et les écrire ...

1. sans les écrire
2. on rappelle que le niveau d'un noeud est la longueur du chemin allant de la racine à ce noeud, les niveaux sont notés à gauche de la figure 2.

Question 6 : Affichage graphique [3 points]

Cette dernière partie concerne l'affichage du réseau dans une fenêtre graphique de largeur L et de hauteur H. Cet affichage consiste à représenter chaque noeud de l'arbre abstrait selon le principe suivant (la figure 1 donne une idée du résultat attendu) :

- on détermine tout d'abord la position (x, y) de chaque noeud : pour le noeud `L.liste[i][j]`, on a $y = (i + 1) * (H/n)$ et $x = (j + 1) * (L/m)$ avec n le nombre de niveau dans l'arbre et m le nombre de noeuds au niveau i ;
- on dessine alors chaque noeud : un triangle "pointe en haut" pour un opérateur *prod*, un demi-cercle pour un opérateur *sigma*, un caractère pour une entrée ;
- on connecte alors ces représentations entre-elles pour finir le dessin du réseau.

Ecrivez le code de la procédure suivante :

```
void afficher (fenetre f ; listeNiveaux L) ;
-- affiche le reseau decrit par L dans la fenetre f
```

On rappelle que la fenêtre est munie d'un repère centré sur son coin supérieur gauche, avec un axe des abscisses horizontal orienté vers la droite et un axe des ordonnées vertical orienté vers le bas. On donne en Annexe E les primitive d'un module `graphsimple` nécessaires à cette question.

Annexe A : extrait du fichier `analyse_lexicale.h`

```
typedef enum  {
   LETTRE,          // lettre minuscule
   PROD,            // l'operateur prod
   SIGMA,           // l'operateur sigma
   PARO,            // (
   PARF,            // )
   VIRG,            // ,
   PVIRG,           // ;
   FIN_SEQ           // lexeme de fin de sequence
} Nature_Lexeme ;

typedef struct {
   nature : Nature_Lexeme;    // nature du lexeme
   lettre : charactere ;      // lettre dans le cas d'un lexeme LETTRE
} Lexeme ;
```

Annexe B : le fichier `arbre_abstrait.h`

```
// definition du type Ast
  #define NMAXFILS 6
  typedef enum  {SIGMA, PROD, LETTRE} TypeAST ;

  typedef struct noeud {
       typeAst nature ;
       char lettre ;   // lorsque la nature est LETTRE
       struct noeud fils[NMAXFILS] ;
       int nb_fils ;
   } Noeud ;

  typedef Noeud *Ast ;
```

```
// definition du type TabSymb
  #define NMAXIDF 10  // le nombre maximale d'entree
  typedef struct {
       char tab[NMAXIDF] ;
       int nb_idf ;
  } TabSymb ;
```

Annexe C : le fichier arbre_abstrait_construction.h

```
   void creer_SIGMA (Ast *A);
   // cree dans *A un noeud SIGMA sans fils

   void creer_PROD (Ast *A);
   // cree dans *A un noeud PROD sans fils

   void creer_LETTRE (char l; Ast *A);
   // cree dans *A un noeud LETTRE de nom l

   void ajouter_fils (Ast *A; Ast F);
   // ajoute l'arbre F comme nouveau fils de l'arbre *A
```

Annexe D : le fichier ensemble.h

```
Ensemble vide();
  // renvoie un ensemble vide (on ne precise pas la definition du type Ensemble)

int appartient (Ensemble E ; char x);
  // vaut vrai ssi x appartient a E

void ajouter (Ensemble *E ; char x);
  // ajoute x a l'ensemble *E (si x est absent de E)

int egalite (Ensemble E1, Ensemble E2);
  // vaut vrai ssi E1 et E2 contiennent les memes elements
```

Annexe E : extrait du fichier graphsimple.h

```
   // Trace une ligne entre les points (x1, y1) et (x2, y2)
   void Ligne(int x1, y1, x2, y2);

   // Trace un demi-cercle a droite du segment defini par les points (x1, y1) et (x2, y2)
   void demiCercle(int x1, y1, x2, y2);

   // Trace un triangle dont les sommets sont les points (x1, y1),
   // (x2, y2) et (x3, y3)
   void Triangle(int x1, y1, x2, y2, x3, y3);

   // Affiche le caractere c au point (x,y)
   void Ecrire(char c ; int x, y);
```